Problem Set 1

1. In this first problem you are asked to define a few arithmetic functions on the natural numbers. We *encourage* you to do so without using any special props, such as Maude's `if_then_else_fi` operator (included together with the built-in `BOOL` module, which is imported by default into any other module), or the `[owise]` feature. We suggest that you not use such props at this point in order to help you become familiar with flexible ways of defining functions in Maude with equations. The typing of the functions you are asked to define is already given. We also include some test results that you should expect to get if these functions are correctly defined. All functions should be familiar to you, except perhaps the *monus* `.-.` operator, which has an easy specification: `N .-. M` is the usual integer difference `N - M` when `N` is greater than or equal to `M`, and `0` otherwise. One last point —which you may find useful when defining some of the predicates below— is that, since Maude's built-in module `BOOL` is automatically imported, besides the sort `Bool`, you also have at your disposal the constants `true` and `false` and the Boolean functions `and`, `or` and `not`.

```
fmod NAT-MIXFIX is
  sort Nat .
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op _+_ : Nat Nat -> Nat .
  op _*_ : Nat Nat -> Nat .
  vars N M : Nat .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
  eq N * 0 = 0 .
  eq N * s(M) = N + (N * M) .
endfm

fmod ARITH-FUNS is protecting NAT-MIXFIX .
  op _^_ : Nat Nat -> Nat .     *** exponentiation
  op fact : Nat -> Nat .        *** factorial
  op _>_ : Nat Nat -> Bool .    *** greater than
  op _.=._ : Nat Nat -> Bool .  *** equality predicate on numbers
  op max : Nat Nat -> Nat .     *** biggest of two numbers
  op min : Nat Nat -> Nat .     *** smallest of two numbers
  op _.-._ : Nat Nat -> Nat .   *** monus
  op |_-_| : Nat Nat -> Nat .   *** absolute value of the integer difference

  vars N M : Nat .

 *** insert here your equations defining each of the above eight functions

endfm

*** Some tests:

red s(s(0)) ^ s(s(0)) .           *** should be s(s(s(s(0))))

red s(s(s(0))) ^ s(s(s(0))) .     ***( should be
s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))))))))))))))))
)

red fact(s(s(s(0)))) .            *** should be s(s(s(s(s(s(0))))))
```

```
red fact(s(s(s(s(0))))) .         ***(should be
s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0))))))))))))))))))))))))
)

red s(s(s(s(0)))) > s(s(0)) .     *** should be true

red s(s(0)) > s(s(s(s(0)))) .     *** should be false

red s(s(0)) .=. s(s(s(s(0)))) .   *** should be false

red s(s(s(0))) .=. s(s(s(0))) .   *** should be true

red max(s(s(0)),s(s(s(s(0))))) .  *** should be s(s(s(s(0))))

red min(s(s(0)),s(s(s(s(0))))) .  *** should be s(s(0))

red s(s(0)) .-. s(s(s(s(0)))) .   *** should be 0

red s(s(s(s(0)))) .-. s(s(0)) .   *** should be s(s(0))

red | s(s(0)) - s(s(s(s(0)))) | . *** should be s(s(0))
```

2. This second problem is about defining some simple list-based functions in Maude. It is also about a couple of other things. A first small point is making you aware of some parsing convenience. The list "cons" operator `_;_` can be made more readable by not having to write extra parentheses. For example, instead of writing `0 ; (s(0) ; (s(s(0)) ; nil))` one would like to just write `0 ; s(0) ; s(s(0)) ; nil`. This can be achieved by giving the Maude parser the "gathering" information `gather (e E)` when declaring `_;_` which instructs it to "right associate" the parentheses when parsing. The second, no so small point, is to make you familiar with ways in which subsorts can be quite powerful, for example to define odd and even natural numbers by declaring subsorts `subsorts Odd Even < Nat .` with appropriate (subsort-overloaded) constructors. You can reap the benefits in this problem by being able to define the functions `oddL` and `evenL`, that respectively return the sublists of odd (resp. even) elements of a list, in a considerably simpler (and more efficient!) way than by explicitly defining `odd` and `even` predicates on natural numbers. Again, we *encourage* you to define all functions without using any special props, such as Maude's `if_then_else_fi` operator or the `[owise]` feature, since this will help you become familiar with flexible ways of defining functions in Maude by equations.

```
fmod NAT-LIST-II is protecting NAT-MIXFIX .
  sorts Odd Even NeList List .
  subsorts Odd Even < Nat .
  subsorts NeList < List .
  op 0 : -> Even [ctor] .
  op s : Even -> Odd [ctor] .
  op s : Odd -> Even [ctor] .
  op nil : -> List [ctor] .
  op _;_ : Nat List -> NeList [ctor gather (e E)] .
  op length : List -> Nat .
  op first : NeList -> Nat .
  op rest : NeList -> List .
  var N : Nat .
  var L : List .
  eq length(nil) = 0 .
  eq length(N ; L) = s(length(L)) .
  eq first(N ; L) = N .
  eq rest(N ; L) = L .
endfm

fmod LIST-FUNS is protecting NAT-LIST-II .
```

```
  op _@_ : List List -> List .        *** list append
  op rev : List -> List .             *** list reverse
  op odd-L : List -> List .           *** sublist of odd numbers
  op even-L : List -> List .          *** sublist of even numbers
  op sigma : List -> Nat .            *** sum of all numbers in the list
                                      *** by convention, sigma(nil) = 0 .

  vars N M : Nat .  var L Q : List .  var O : Odd .  var E : Even .

  *** insert here your equations for each of the above five functions

endfm

*** Some tests:

red (0 ; s(0) ; s(s(0)) ; nil) @ (s(0) ; s(s(0)) ; s(s(s(0))) ; nil) .    ***(should be

0 ; s(0) ; s(s(0)) ; s(0) ; s(s(0)) ; s(s(s(0))) ; nil
)

red rev(0 ; s(0) ; s(s(0)) ; nil) .  *** should be s(s(0)) ; s(0) ; 0 ; nil

red odd-L(0 ; s(0) ; s(s(0)) ; s(0) ; s(s(0)) ; s(s(s(0))) ; nil) .  ***( should be

s(0) ; s(0) ; s(s(s(0))) ; nil
)

red even-L(0 ; s(0) ; s(s(0)) ; s(0) ; s(s(0)) ; s(s(s(0))) ; nil) . ***( should be

0 ; s(s(0)) ; s(s(0)) ; nil
)

red sigma(0 ; s(0) ; s(s(0)) ; s(0) ; s(s(0)) ; s(s(s(0))) ; nil) .  ***( should be

s(s(s(s(s(s(s(s(s(0)))))))))

)
```